Ecole Polytechnique de l'Université François Rabelais de Tours
Département Informatique
64 avenue Jean Portalis
37200 Tours, France
Tél. +33 (0)2 47 36 14 14
polytech.univ-tours.fr

Rapport de stage

2018-2019

Création d'un crawler XMLTV

Confidentiel

Entreprise
Université de Tours

Tuteur entreprise
Mathieu Delalandre (Maître de conférence)

Étudiant
Romain Hérault (DI4)

Tuteur académique
Vincent T'Kindt

22 août 2019

Liste des intervenants

Entreprise

Université de Tours
60 rue du Plat d'étain
37000 Tours
www.univ-tours.fr

Nom	Email	Qualité
Romain Hérault	romain.herault-2@etu.univ-tours.fr	Étudiant DI4
Vincent T'Kindt	vincent.tkindt@univ-tours.fr	Tuteur académique, Département Informatique
Mathieu Delalandre	mathieu.delalandre@univ-tours.fr	Tuteur entreprise, Maître de conférence

Avertissement

Ce document a été rédigé par Romain Hérault susnommé l'auteur.

L'entreprise Université de Tours est représentée par Mathieu Delalandre susnommé le tuteur entreprise.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Vincent T'Kindt susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnait assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnait qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.

Pour citer ce document

Romain Hérault, *Création d'un crawler XMLTV*, Rapport de stage, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

```
@mastersthesis{
    author={Hérault, Romain},
    title={Création d'un crawler XMLTV},
    type={Rapport de stage},
    school={Ecole Polytechnique de l'Université François Rabelais de Tours},
    address={Tours, France},
    year={2018-2019}
}
```

Document mis en forme sous LaTeX
grâce à la classe de document `polytech.cls` V2.2.5 (2017/05/02)

Table des matières

Table des figures

Liste des tableaux

Introduction

1 L'entreprise

Todd.tv est un projet de start-up créé par Mathieu Delalandre, chercheur en informatique avec une carrière internationale, et Jordan Nicot, ingénieur informatique. Celui-ci répond à plusieurs problématiques telles que l'explosion du nombre d'opérateurs et du panel de choix de ces derniers en ce qui concerne les dizaines de chaînes de télévision et les plate-formes de Vidéo à la Demande (VoD, Replay ou Streaming), ou encore le changement de comportement des utilisateurs de la télévision qui sont aujourd'hui "dual-screener". Ils utilisent les mobiles et tablettes tout en regardant la TV ou la VoD pour la recherche d'informations, de contenu additionnel, pour accéder à des réseaux sociaux en lien avec leur programme etc.

Todd.tv va leur proposer de nouveaux services permettant le rapprochement entre la TV, la VoD, le digital et internet. La start-up va créer une plateforme qui oeuvrera comme un guide TV/VoD 3.0 auprès des utilisateurs. Ce sera un guide ergonomique, pensé pour les utilisateurs, où la recherche deviendra plus simple, plus courte et pourvue de contenus de qualité en fonction des différents profils utilisateurs.

De plus Jordan et Mathieu misent sur l'ajout de fonctionnalités telles que les détections des plages et des spots publicitaires afin de proposer une alternative à ceux qui souhaitent passer la publicité à la télévision, ainsi que du drive-to-web.

Le projet est engagé au sein de l'Université de Tours depuis octobre 2017 et ce pour une durée de 3 ans. Il a fait l'objet d'une étude de marché, de la rédaction d'un business plan et d'un rapprochement auprès de différentes structures d'accompagnement de la Région Centre-Val de Loire. L'entrée en phase de maturation du projet et la prévision de la création de l'entreprise sont prévues courant 2020-2021.

2 Le projet

Les résultats des études de marché engagées avant notre arrivée en période de stage ont permis à Mathieu et Jordan de définir le périmètre du projet et d'isoler les fonctionnalités à proposer aux utilisateurs.

L'application agit comme une passerelle aux contenus vidéo. Les programmes sont exclusifs aux plate-formes qui les diffusent, et à un instant donné il est difficile de choisir un programme

indépendamment de sa source. On regarde un guide TV pour choisir un programme à la télévision et on se connecte, par exemple à Netflix, pour voir les programmes disponibles. Par rapport à la télévision traditionnelle, un des objectifs est d'informer le téléspectateur précisément de la diffusion des programmes en temps-réel, et pas seulement sur des plages temporelles larges, comme c'est actuellement le cas. Cela se traduit par l'identification des temps publicitaires, des démarrages et des reprises des programmes. On parle d'une d'assistance télévisuelle, qui s'inscrit dans le contexte de centralisation des contenus vidéo. A l'image des programmes de VoD démarrés par l'utilisateur, on cherche à donner plus de contrôle à l'utilisateur lorsqu'il regarde la télévision. Ainsi, comme le montre l'étude OpinionWay, les français sont majoritairement actifs pendant les coupures publicitaires, on pourrait donc imaginer leur faire bénéficier d'une alarme ou d'un retour automatique vers leur programme lorsqu'il reprend.

Nous pouvons donc découper l'application en trois parties distinctes (voir Figure 1). Les parties station et back-end sont charger de récupérer des informations et de les transmettre à la partie front-end.

Station

La station est la partie de l'application qui s'occupe d'observer en direct les flux TV et de détecter par exemple les pubs, ou bien les différents programmes. C'est grâce à elle que l'utilisateur pourra savoir si une chaîne est en train de diffuser de la publicité.

Back-end

La partie back-end s'occupe de collecter les informations nécessaires à l'élaboration du guide TV amélioré.

Front-end

La partie front-end s'occupe de l'affichage et de toutes les interactions avec l'utilisateur.

Chacune de ces parties a été confiée à un stagiaire. Mon travail a donc été centré sur la partie back-end.

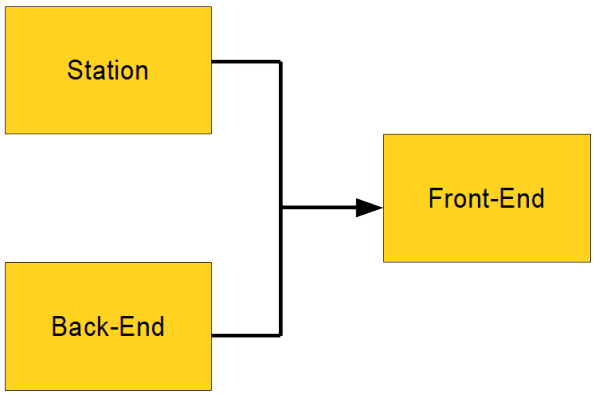

Figure 1 – *Structure globale de l'application*

3 Les objectifs du stage

Initialement, plusieurs objectifs étaient définis :

— Mise en œuvre du crawler Louis, maintien de la base de données images,
— Mise en place du crawler XMLTV, configuration,
— Ouverture / configuration serveur LIFAT,
— Implémentation des parseurs XMLTV / Json,
— Mise en place de la structure moniteur,
— Synchronisation FTP front-end,
— Extension collecte image, formatage,
— Tests mémoire, mise en place watchdog,
— Thread monitoring,

Le but était d'essayer de faire un maximum de ces tâches avant la fin du stage, tout en sachant qu'elles ne seront probablement pas tous réalisées.

On peut voir qu'une grosse partie de ces tâches est liée au développement logiciel (crawler XMLTV, parseurs, structure moniteur, synchronisation FTP, collecte d'image et formatage, tests mémoire, watchdogs, thread monitoring), mais quelques unes ne le sont pas (mise en oeuvre du crawler Louis, maintien de la BDD image, et ouverture et configuration du serveur LIFAT).

1 Éléments extérieurs au développement

La grande majorité de mon stage a été du développement d'application. Cependant, j'ai aussi eu quelques tâches a effectuer qui n'avaient rien à voir avec de la programmation.

1 Mise en œuvre crawler Louis, maintien base de données images

1.1 Le crawler de Louis

Description du projet

Durant l'année, un projet de recherche et développement a été effectué par Louis Babuchon. Le but de son projet était entre autre de réaliser une base de données d'images recueillies sur différents sites internet de guide TV. Il a donc conçu un logiciel en Java qui permet, en suivant une liste de sites, de récupérer les images concernant tous les programmes TV et de les stocker. PLus d'informations sont disponibles dans son rapport (voir [1]).

Intérêt

Ce programme peut s'avérer utile pour la start-up Todd.tv. En effet, un des objectifs finaux de l'application de l'entreprise est de pouvoir avoir plusieurs images pour chacun des programmes présents sur le guide. Le logiciel de Louis peut donc nous servir à créer une base de données suffisamment fournie pour être intégrée à l'application.

Les données

Chaque image est stockée dans un dossier correspondant à un programme TV. Bien évidement, il arrive qu'il y ait de la redondance entre ces images, plusieurs sites utilisent souvent des mêmes images (ou alors des versions légèrement modifiées). Le programme de Louis s'occupe donc aussi de détecter les doublons pour ne garder que la meilleure version.

1.2 Utilisation du crawler

Ma mission

Toutes ces données étant récupérées directement sur les sites internet, il faut lancer l'application régulièrement pour récupérer les nouvelles images et les nouveaux programmes présents sur ces sites. Une de mes missions pendant ce stage a donc été de relancer régulièrement l'application de Louis (environ toutes les semaines), pour continuer à alimenter la base d'images. La quantité d'image récupérée dépend fortement de la dernière date à laquelle on a lancé le programme. Par exemple, la première fois que je l'ai lancé (soit quelques mois depuis sa dernière mise à jour), le programme a récupéré 5.34 Go de données, alors que lorsque je l'ai relancé 3 jours plus tard, je n'ai eu que 300 Mo de nouvelles images.

Lancement du programme

L'utilisation du programme de Louis est assez simple puisqu'il s'agit d'une application java. Sous Linux, il suffit d'exécuter la commande `java -jar Crawler.jar config.json`. Sous Windows, il y a même un script Run.bat, il suffit donc de cliquer sur celui-ci pour lancer automatiquement le programme. Dans la console, nous pouvons voir que l'exécution se déroule en deux parties, la première consiste à récupérer tous les liens menant vers des images que l'on peut trouver sur différents sites, et la deuxième s'occupe de sauvegarder les images. Il peut y avoir un certain temps d'attente (plus de 5 minutes) entre ces deux parties où rien n'est écrit dans la console. Ce temps correspond en fait au téléchargement des images ainsi qu'à leur traitement (on enlève les images que l'on a déjà). Comme cette partie peut être longue, il faut patiemment attendre la fin de l'exécution du programme.

Organisation des données

Les images sont stockées directement dans des dossiers. L'organisation est assez simple, chaque image est stockée dans un dossier dont le nom correspond au programme TV de l'image. Ainsi on se retrouve avec autant de dossiers qu'il y a de programmes différents, et dans chacun de ces dossiers, une ou plusieurs images correspondant au programme.

Fichier de configuration

Un fichier de configuration au format JSON est utilisé pour pouvoir paramétrer les crawlers. Plusieurs paramètres sont disponibles pour chaque crawler. Les paramètres importants sont les suivants :

- Pour le crawler XMLTV,
  - url : le lien de la page d'accueil du site sur lequel on récupère le fichier XMLTV,
  - xml_url : le lien vers le fichier xmltv.xml.
- Pour les crawlers liés au site des guides TV,
  - timeout : le temps que le thread passe à chercher les programmes.
- Pour le downloader,
  - image_path : l'emplacement du dossier où seront stockés toutes nos images. Il est possible de mettre un dossier où l'on a déjà des images, elles seront alors complétées (elles ne seront pas effacées ou écrasées).

Contraintes

Le crawler de Louis étant multi-threadé et récupérant un grand nombre d'images sur plusieurs sites différents, il est nécessaire d'avoir un très bonne connexion internet pour l'exécuter. L'usage d'une connexion fibre avec un bon débit descendant permet d'obtenir de bons résultats rapidement.

Etat de la base de données

Voir l'image Table 1 pour connaître les derniers chiffres concernant la taille de la base de données.

Table 1 – *Contenu de la base de données du crawler de Louis*

Nombre d'images	Nombre de programmes	Taille (en Go)
98 612	38 408	13,4

2 Le serveur

La partie back-end à développer doit être exécutée sur un serveur. Le serveur final de l'application sera sûrement un serveur hébergé chez un fournisseur, mais pour le moment, nous devions utiliser le serveur du LIFAT. Une de mes missions était donc l'ouverture et la configuration de ce serveur.

Spécifications du serveur

Le serveur que nous avons demandé est un serveur à 8 coeurs et 16 Go de mémoire RAM, avec un stockage HDD de 512 Go. Le système d'exploitation est Windows 10.

Connexion au serveur

Nous avons donc accès à une machine virtuelle sur le serveur. Le moyen le plus simple pour s'y connecter est, sur les machines de Polytech, d'utiliser la connexion Bureau à distance (TSE) de Windows. Voici les informations de connexion :

— IP : 10.108.99.58
— Login : Administrateur
— Password : Todd.tv6

Estimation des coûts futurs

J'ai aussi fait une estimation des coûts pour la location d'un serveur hébergé par un autre fournisseur. En effet, l'entreprise n'aura pas accès indéfiniment au serveur du LIFAT, alors il est important d'estimer les coûts d'une solution externe.

Aprés comparaisons de plusieurs solutions, il ressortait que le prix moyen serait d'environ 60€ par mois, soit 720€ par an. De plus les serveurs proposés sont en général plus performants que celui du LIFAT (aussi bien en terme de puissance de calcul que de mémoire vive ou de stockage).

2 L'application

Comme nous l'avons vu dans l'introduction, je me suis occupé de la partie back-end de l'application. L'objectif est de pouvoir fournir toutes les informations nécessaires à la partie front-end pour pouvoir faire un guide TV enrichi de toutes les chaînes françaises. L'idée est qu'en plus de la programmation, le guide puisse afficher des informations supplémentaires sur les programmes, comme par exemple des news sur les acteurs, ou bien des images supplémentaires.

J'ai donc développé un programme en Java pour répondre à ces besoins. Pour ce faire, notre programme est divisé en plusieurs parties. Tout d'abord un premier crawler s'occupe de télécharger régulièrement toutes les informations relatives à la programmation TV. Ces informations sont stockées sous la forme d'un fichier XML, il faut donc un parseur qui va s'occuper de formater les informations et de séparer chacun des programmes. Les programmes et les chaînes seront alors stockés individuellement dans des fichiers XML. Ils pourront alors être enrichis par d'autres crawlers. Ensuite Le programme va convertir les chaînes et les programmes voulus en fichiers JSON qui seront envoyés au front-end.

La norme XMLTV

Les informations concernant les futurs programmes TV qui vont passer à la télévision sont stockées sur un fichier XML particulier, appelé XMLTV. Il s'agit de la norme utilisée pour stocker des informations relatives aux programmes télévisés. Les balises utilisées par cette norme permettent de décrire soit une chaîne, soit un programme.

Voici un extrait d'un fichier XMLTV (les liens URL ont été raccourcis) :

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE tv SYSTEM "xmltv.dtd">

<tv source-info-url="https://api.telerama.fr" source-data-url="https://api.telerama.fr" ↩
        generator-info-name="XMLTV France" generator-info-url="https://www.xmltv.fr">
 <channel id="C111.api.telerama.fr">
    <display-name>Arte</display-name>
    <icon src="https://television.telerama.fr/sites/.../tv/500x500/111.png" />
  </channel>
   <channel id="C118.api.telerama.fr">
    <display-name>M6</display-name>
    <icon src="https://television.telerama.fr/sites/.../tv/500x500/118.png" />
  </channel>
  <programme start="20190707131000 +0200" stop="20190707133000 +0200" showview="" ↩
          channel="C294.api.telerama.fr">
```

```
14      <title>Par ici les sorties</title>
15      <sub-title>Episode 42/52</sub-title>
16      <desc lang="fr">Le point sur les sorties de la semaine.</desc>
17      <category lang="fr">magazine du cinéma</category>
18      <length units="minutes">20</length>
19      <icon src="https://television.telerama.fr/...90ca-e76da685164e.jpg" />
20      <video>
21        <aspect>4:3</aspect>
22        <quality>HDTV</quality>
23      </video>
24      <audio>
25        <stereo>stereo</stereo>
26      </audio>
27      <previously-shown />
28      <rating system="CSA">
29        <value>Tout public</value>
30      </rating>
31    </programme>
32  </tv>
```

Dans cet exemple, on peut voir les deux chaînes Arte et M6 décrites dans des balises channel et le programme "Par ici les sorties" décrit dans une balise programme.

On peut récupérer assez facilement ce fichier sur internet. Il est par exemple disponible pour les programmes français sur le site xmltv.fr et est régulièrement mis à jour.

0.1 Mise en place crawler XMLTV, configuration

Cette partie traite du premier crawler, celui permettant des récupérer les informations de programmation TV.

Crawler

Un crawwler, ou robot d'indexation, est un programme qui se charge de parcourir le web dans le but de récupérer des données. Les crawlers sont notamment utilisés pour l'indéxation des moteurs de recherche, la création de bases de données spécialisées ou bien dans les comparateurs de prix. Normalement, un crawler parcourt le web de manière récursive à partir d'une page de départ et continue sont exploration en suivant différentes règles (voir [2]). Dans notre cas, il s'agit simplement de récupérer régulièrement un fichier à partir d'un lien, donc il ne s'agit pas d'un crawler à proprement parler. Cependant j'ai décider de garder ce terme pour la suite. En revanche, une utilisation de crawlers pourra être faite dans le futur pour l'enrichissement des données (voir Section 1 (Chapitre 5)).

Ce crawler a pour but de récupérer le fichier XMLTV sur le site xmltv.fr. xmltv.fr est un site qui regroupe les informations disponibles en open source sur les programmes tv (en l'occurence celles de Télérama) pour nous proposer un fichier XMLTV exploitable. Ce site permet donc de télécharger gratuitement et facilement le fichier XMLTV avec toutes les informations concernant la programmation TV. Le site a cependant l'air d'être une initiative personnelle, ce qui veut dire qu'il peut être sujet à des pannes ou des bugs. Il peut par exemple y avoir quelques erreurs dans le fichier, comme par exemple des caractères spéciaux interdits ou des liens invalides (en httpss). De il n'y a aucune garanti concernant la robustesse du site, qui pourra cesser d'exister du jour au lendemain. Si cela arrive, il faudra alors se rabattre sur une autre source de XMLTV comme par exemple sur https ://allfrtv.ga/xmltv.php ou http ://xmltvfree.free.fr/.

Le crawler doit être configurable en utilisant un fichier de configuration en XML. Plusieurs paramètres sont à prendre en compte.

Le temps de rafraîchissement

Le crawler va régulièrement essayer de télécharger le fichier XMLTV. Pour définir la fréquence de la requête, il faut modifier un paramètre appelé "refreshTime" dans le fichier de configuration. Ce paramètre est exprimé en minutes et correspond à la période de temps entre deux requêtes de téléchargement. Le fichier XMLTV n'étant actualisé que tous les jours sur internet, il n'est pas nécessaire de fixer un refreshTime trop petit. Par exemple un refreshTime de 12h peut suffire.

Le mode forcé

Le fichier XMLTV est un fichier assez lourd, il pèse plus de 80 Mo. Il faut donc essayer de limiter les téléchargements lorsqu'ils ne sont pas nécessaires. Pour cela, avant chaque téléchargement, le crawler vérifie sur le site xmltv.fr la date de la dernière mise à jour du fichier. Si cette date est plus récente que celle enregistrée (et qui correspond à la dernière fois que l'on a téléchargé le fichier), le crawler télécharge le nouveau fichier. Sinon il se remet en attente.

Pour des besoins de test, le mode forcé à été mis en place. Il s'agit d'un mode où les dates de mises à jour ne sont plus prises en compte et où le crawler va forcement lancer le téléchargement à chacun de ses réveils. Ce mode peut être activé dans le fichier de configuration, il suffit de passer le paramètre "forceRecovery" à true.

0.2 Parseur XMLTV

Un fois que le crawler a effectué sont travail, nous nous retrouvons avec un fichier assez conséquent et contenant plein d'informations. L'étape suivante est donc d'extraire ces informations, et de les stocker sous une forme plus facilement exploitable. Ce travail est effectué par le parseur XMLTV.

Le but du parseur est de convertir les différents éléments du XMLTV en objets Java que nous pourrons par la suite réutiliser.

Le parseur XMLTV doit répondre à plusieurs problématiques. Du fait de la taille du fichier XMLTV, il doit être le plus efficace possible pour que le parsing s'effectue assez rapidement. De plus il faut essayer de trouver une solution pour éviter de charger tout le fichier d'un coup en mémoire.

0.2.1 Parseur SAX

En Java, il possible de créer un parseur XML de plusieurs façons, dont les deux principales sont le DOM et le SAX. Nous avons choisis d'utiliser le parseur SAX, car il est plus rapide et plus efficace. De plus, ce parseur fonctionne par évènement, c'est-à-dire qu'il appelle les fonctions de parsing à certain moments bien précis. Pour être plus clair, le parseur parcourt le fichier XML séquentiellement, et lorsqu'il arrive au début d'une balise, il va appeler la fonction startElement, de même il va appeler la fonction endElement lorsqu'il arrive à la fin d'une balise. L'avantage de cette façon de faire est que l'on ne charge pas tout le document en mémoire, ce qui permet de gagner en efficacité (voir [3]).

0.2.2 Les objets du modèle

Le parseur nous permet de convertir les éléments du XMLTV en objets. Deux types d'éléments sont représentés dans ce fichier, les programmes et les chaînes. Nous avons donc créer l'équivalent de ces éléments en objets Java, respectivement Programme et Channel. Ces objets

contiennent exactement les informations présentes dans le XMLTV sous forme de String. Ce ne sont que des objets temporaires puisque l'objectif est de les retransformer en fichier XML juste après. Pour cela, ces deux objets ont une méthode toXML() qui permet de retranscrire toutes leurs informations en un unique String au format XML.

A noter qu'il y a une troisième classe dans le package model. Il s'agit de Credit, qui représente les personnes composant le casting d'un programme, avec leur rôle (présentateur, réalisateur, acteur...). Chaque Programme contient donc un objet Credit contenant les membres de son casting.

0.3 Les fichiers

Tout le contenu parsé du XMLTV est stocké sur le serveur. Nous avons choisi d'utiliser du NoSQL pour stocker les données, nos programmes et nos chaînes sont donc stockés sous forme de fichiers XML. Chaque fichier correspond à un programme ou à une chaîne. Cela permet de garder tous les programmes sauvegardés et ainsi, en parallèle, d'autres crawlers peuvent venir ajouter des informations, comme des chemins vers d'autres images ou bien des données supplémentaires concernant par exemple des news sur les acteurs etc... Ainsi faire une base de données aurait été plus laborieux notamment pour l'ajout d'images et n'était pas nécessaire étant donné que nous avions déjà un modèle relationnel métier transcris via la DTD du XMLTV.

Identification des fichiers

Il a aussi fallu trouver un moyen de nommer chacun de ces fichiers.

Pour les chaînes, cela ne pose pas de problème car chaque chaîne a déjà un id unique qui lui est attribué dans le XMLTV, il suffit donc de prendre cet id comme nom de fichier.

Pour les programmes, c'est un peu plus complexe. Une manière que l'on a trouvé pour faire ceci est d'attribuer un id unique en fonction du contenu de chaque fichier. En fait c'est dans l'objet Programme dont nous avons parlé précédemment que nous faisons cela. En Java, il est possible d'implémenter une fonction hashcode() qui permet de renvoyer un entier signé unique en fonction des caractéristiques de l'objet. Nous avons donc implémenté cette fonction pour notre objet Programme ainsi que tous les éléments le composant. Cela fait que dès que deux programmes ont un paramètre différent, ils auront alors normalement des ids différents. Il suffit ensuite d'inscrire cet id dans les fichiers XML correspondant aux programmes et de nommer chaque fichier en fonction de cet id.

La fonction de hashcode fonctionne selon le principe suivant :

- On choisi un nombre premier (dans notre cas il s'agit de 31)
- On calcule le hashcode du premier paramètre de l'objet (comme les paramètre ne sont que des string, il s'agit du hashcode d'une string qui est calculé de manière similaire avec chacun de ses caractères), on ajoute 1 a ce résultat et on multiplie le tout par notre nombre premier.
- On prend le hash du paramètre suivant que l'on ajoute au résultat et on multiplie le tout par le nombre premier.
- On ré-itère l'opération pour tous les paramètres. Le nombre obtenu à la fin est notre hashcode.

Attention cependant, les hashcodes ne garantissent pas l'unicité des ids. En effet, il est tout de même possible que deux programmes complètement différents se retrouvent avec le même hashcode. Cependant, comme notre plage d'ids différents est très large et que nous calculons le hashcode sur des objets assez complexes, la probabilité que deux objets différents aient le même hashcode est très faible.

0.4 Optimisations

La partie parsing de l'application est relativement lente. Pour découper l'ensemble du XMLTV, on peut en avoir pour plus de 7 minutes. Cela n'est pas forcément gênant dans le sens où l'application ne va effectuer ces opérations qu'une seule fois par jour, mais il peut quand même être intéressant de chercher à optimiser l'algorithme. Pour cela, plusieurs méthodes ont été testées.

On peut noter que tous les tests ont été effectués avec un SSD, que l'on peut déjà considérer comme une optimisation matérielle par rapport aux disques durs classiques.

0.4.1 Optimisation grâce aux ids

Ce n'est pas le parsage en lui-même qui prend le plus de temps, mais l'écriture des nombreux fichiers XML. En effet, après test, nous nous sommes rendu compte le parsage en lui-même du XMLTV est très rapide (2 secondes environ), alors que l'écriture des fichiers prend jusqu'à 7 minutes.

Un des moyens pour gagner du temps est donc d'essayer de limiter le nombre de fichiers à écrire. Pour cela, nous pouvons utiliser le système d'ids vu précédemment . En effet lorsque l'on réimporte notre XMLTV, on le parse et on obtient nos nouveaux programmes. Il suffit ensuite de voir si l'id du nouveau programme est déjà dans la base de donnée, et si il y est déjà, il n'est alors pas nécessaire de l'ajouter.

Pour faire cela rapidement, dès que l'on parse notre XMLTV, on sauvegarde tous les ids dans un tableau, que l'on sérialise. Ainsi au prochain parsage, on peut comparer l'ancien tableau d'ids avec le nouveau. On ne touche pas au fichiers dont les ids sont dans les deux tableaux, on ajoute ceux qui ne sont que dans celui des nouveaux ids, et on supprime ceux qui ne sont que dans le tableau des anciens ids (voir figure Figure 1).

Cela a aussi un autre avantage, c'est que si entre temps, un crawler a ajouté des informations à un fichier, elles ne seront pas écrasées puisque si le programme n'a pas changé avec le nouvel XMLTV, on va garder le même fichier pour ce programme. De plus, si un programme a été modifié dans le XMLTV, alors il n'aura plus le même id et sera remplacé correctement.

Cette méthode nous permet de gagner du temps surtout si l'on télécharge le XMLTV régulièrement. Entre deux jours le XMLTV ne change pas beaucoup, et donc il y a seulement quelques changement à effectuer dans la base de données. Forcement, cette méthode sera inefficace si l'on lance le crawler avec deux mois d'intervalle, car les programmes seront alors tous différents. Pour notre utilisation (téléchargement du XMLTV tous les jours), cette méthode est efficace car il a moins d'un quart des programmes à changer.

En évitant de réécrire tous les fichiers existants, et en supprimant les anciens qui ne sont pas présents dans le nouvel XMLTV, on réduit le temps d'écriture à environ 2 min 30.

Le problème avec cette méthode est qu'il dépend de la manière dont nous avons choisis les ids (hashcode des informations des programmes). Si l'on trouve une nouvelle manière d'identifier nos programmes, alors nous ne pourront plus l'utiliser.

0.4.2 Multithreading

Un autre solution consiste à utiliser le multi-threading. En effet, le parsage du XMLTV étant très rapide, on peut essayer de gagner du temps sur la création et l'écriture des fichiers XML en la

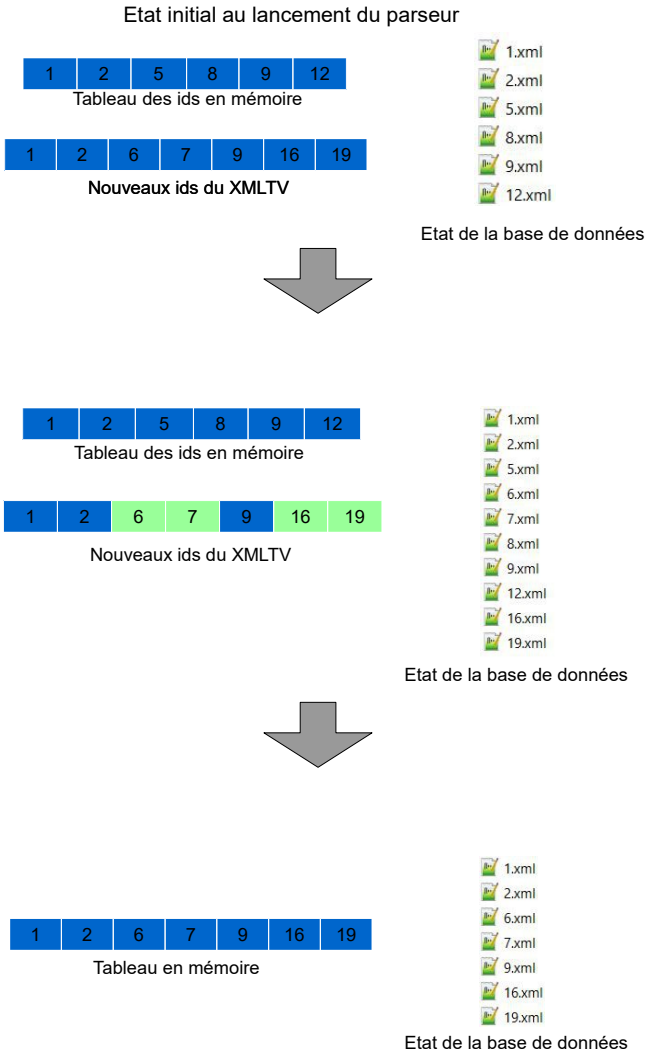

Figure 1 – *Fonctionnement de l'optimisation grâce aux ids*

faisant parallèlement. Comme chaque programme est écrit dans un fichier qui lui est propre, il n'y a pas de problème de cohérence des données. On peut donc définir une tâche à effectuer par un thread comme étant l'écriture de notre objet Programme en XML dans un fichier et utiliser un executor en Java qui va se charger de distribuer ces tâches à un pool de threads de taille prédéfini. Le problème est que si l'on donne les tâches à l'executor dès qu'on les a parsées, on va vite se retrouver à utiliser beaucoup de mémoire.

En effet, comme le parsing est beaucoup plus rapide que l'écriture des fichiers, on va vite finir de parser le fichier et se retrouver avec plus de 80000 tâches en attente d'être exécutées. Pour palier à ce problème, il suffit de stocker un certain nombre de ces tâches (disons 500) dans un buffer, et lorsque le buffer est plein, le parser s'arrête et attend qu'elles soient exécutées avant de reprendre. Cela augmente très légèrement le temps d'exécution mais permet de diminuer grandement la mémoire utilisée. De nouveaux paramètres ont donc été rajoutés dans le fichier de configuration du crawler, il s'agit du nombre de threads alloués (sous parser.mutlithreading.threadsNumber)

et du nombre de tâches maximum dans le buffer (sous parser.multithreading.tasksNumber). Après plusieurs tests, ces paramètres ont été respectivement fixés à 16 et 512, mais ces nombres dépendent du nombre de coeurs de la machine utilisée et de la mémoire que l'on veut occuper ainsi que des performances d'écriture sur le disque (si le disque est très performant alors on peut diminuer le nombre de tâches).

Le multithreading nous permet de gagner à peu près autant de temps que l'optimisation grâce aux ids, c'est-à-dire que l'on passe d'environ 7 min à 2 min 30.

0.4.3 Filtrage des chaînes

Une autre méthode d'optimisation qui à la base n'a pas été pensée pour ça, est le filtrage des chaînes. Il se trouve que du côté front-end, afficher tous les programmes et toutes les chaînes était très lourd. Non seulement en terme de calculs, mais aussi d'interface. On peut facilement comprendre qu'afficher une liste de plus de 600 chaînes différentes est assez peu utile pour le l'utilisateur. Il a donc été fait comme choix de se limiter seulement aux chaînes de la TNT, ce qui réduit le nombre de chaînes à une trentaine et donc le nombre de programmes (qui passe d'environ 100 000 à moins de 10 000).

Pour faire cela, des nouveaux champs ont été ajoutés dans le fichier de configuration, il s'agit des champs sous parser.filter.authorizedChannels. A cet endroit il suffit d'ajouter une balise channelId contenant l'id de la chaîne que l'on veut garder pour qu'elle soit prise en compte. Toutes les chaînes dont l'id n'est pas dans cette liste seront donc ignorées, tout comme les programmes diffusés par cette chaîne. Il est aussi possible de désactiver le filtrage des chaînes en mettant le champ parser.filter.enabled à false.

Comme on peut l'imaginer, réduire le nombre de chaînes et de programmes réduit aussi le temps de traitement du programme, donc on peut considérer que c'est une optimisation.

0.4.4 Méthodes utilisées

Puisque toutes les méthodes vues précédemment sont compatibles, nous les utilisons donc toute dans le but d'optimiser au mieux le temps de calcul.

En utilisant aucune de ces méthodes, donc sur plus de 600 chaînes et environ 100 000 programmes, en réécrivant tous les programmes dans la base de données, et sans utiliser de multi-threading, nous avons vu que nous devons attendre environ 7 minutes.

En les utilisant toutes les méthodes d'optimisation, c'est-à-dire en ne travaillant qu'avec une trentaine de chaînes, soit moins de 10 000 programmes, en ne réécrivant pas le programmes que l'on a déjà, et avec du multi-threading, le parsing dure alors 6 secondes (sachant que le test à été effectué avec une base de données vieille de 2 jours).

1 Le stockage

Nous avons donc vu comment sont créés les fichiers XML et ce qu'ils contiennent. Nous allons maintenant voir comment accéder aux fichiers et aux objets qu'il contiennent, ainsi que les méthodes mises en place pour accélérer les temps d'accès.

1.1 Problématique

Stocker nos fichiers XML "en vrac" dans des dossiers soulève la problématique des temps d'accès. En effet, imaginons que l'on veuille accéder à tous les programmes TV ayant un certain titre, alors nous sommes obligé de prendre chacun des fichiers XML dans notre dossier programme, de l'ouvrir, de parser son contenu sous forme d'objet, et de vérifier si le champ titre correspond bien à celui que l'on recherche.Lorsque le nombre de programmes est grand (et rappelons que l'on peut atteindre 100 000 programmes) cette opération prend énormément de temps. Il faut aussi prendre en compte le fait que le but de cette base de données est que des crawlers accèdent à ces fichiers pour y ajouter des informations, ils auront donc besoin de récupérer les fichiers qui les intéressent, et ce le plus rapidement possible car ces crawlers vont devoir compléter le maximum de fichiers le plus rapidement possible.

Nous avons donc besoin d'optimiser les opérations d'accès aux fichiers pour les rendre plus performantes. L'idée est de ne pas avoir à ouvrir et parser les fichiers, et même de ne pas avoir à étudier tous les fichiers.

1.2 MultiHashMap

Pour ce faire, nous avons utilisé des MultiHashMaps. Il s'agit de HashMap qui peuvent contenir plusieurs valeurs pour une même clé. On peut donc par exemple prendre le titre du programme comme clé, et pour la valeur qu'elle référence, on y stocke l'objet File représentant le fichier. Ainsi, lorsque l'on a besoin de tous les programme dont le titre est par exemple "Journal", il suffit de rechercher dans la hashMap les valeurs dont la clé est "Journal". La map va alors nous retourner un objet List contenant les fichiers dont le titre est "Journal".

Cette technique nous permet ainsi de récupérer les objets voulus dans un temps très court. Par exemple, si nous voulons récupérer les fichiers par titre avec la méthode expliquée dans la sous section précédente, avec une recherche parmi environ 100000, cela prendrait environ 14 secondes, peut importe le nombre de programmes ayant ce titre. Avec les hashMaps, le résultat dépend du nombre de programmes ayant ce titre, mais cela varie entre 1 milliseconde et 40 millisecondes, respectivement pour 1 et 250 programmes ayant le titre recherché.

Il ne suffit ensuite plus qu'à créer une map par élément dont on a besoin pour rechercher un fichier. Par exemple, les maps en fonction du titre, de la journée de diffusion, et des membres du casting sont déjà implémentées.

Lorsque l'on a fini d'utiliser les maps, elles sont sérialisées pour pouvoir les réutilisées lorsque l'on relance le programme.

L'ajout d'une nouvelle map, ainsi que l'accès aux fichiers se fait au travers de la classe Database.

1.3 La classe Database

La classe Database est une classe contenant des méthodes statiques dont l'objectif est de pouvoir avoir accès aux fichiers contenant les programmes et les chaînes facilement. C'est aussi dans cette classe que l'on créer les différentes Hashmaps qui nous servent à accéder aux fichiers plus rapidement.

Parmi les méthodes de cette classes se trouvent celles qui permettent d'avoir accès à tous les fichiers des chaînes ou des programmes, ainsi qu'aux fichiers stockés dans les hashMaps.

1.3.1 Ajout d'une hashMap

Si, pour les besoins d'un crawler, il est nécessaire d'ajouter une hashmap sur un élément particulier d'un programme, voici comment faire.

- 1) Ajouter une nouvelle SpecializedHashMap en tant que variable de classe, et lui donner un nom (de préférence en fonction de la clé utilisée).
- 2) Dans la méthode buildNewStorage : Construire la SpecializedHashMap, dont le type est celui de la clé . L'ajouter à l'ArrayList maps.
- 3) Ajouter une méthode addProgrammeToXXXMap qui retourne l'ArrayList d'éléments extraits à partir de l'objet programme (si il n'y a qu'un élément, le mettre dans un ArrayList de taille 1).
- 4) Ajouter cette méthode à la méthode addProgrammeToMaps.
- 5) Créer la méthode getProgrammeByXXX retournant le tableau des fichiers stockés dans la map en fonction de la clé.
- 6) Si nécessaire ajouter la méthode correspondante dans ObjectAccess (voir Section 1.4.1).

Les méthodes de sauvegarde et de vidage sont automatiquement appelées grâce à l'ajout dans l'ArrayList maps.

1.4 La classe ObjectAccess

La classe ObjetAccess est la classe qui contient les méthodes pour accéder non pas aux fichiers des programmes et des chaînes, mais aux objets en eux-mêmes. Ses méthodes (statiques) s'occupent simplement de récupérer les fichiers grâces aux méthodes de la classe Database et de les parser pour récupérer les objets nécessaires.

1.4.1 Ajout d'une HashMap

Si une HashMap est ajoutée dans la classe Database, il peut être nécessaire d'ajouter la méthode correspondante dans la classe ObjectAccess. Pour cela, il suffit d'ajouter la méthode getProgrammeByXXX, qui renvoie un ArrayList de Programmes, et qui prend en paramètre la clé. Le contenu de la méthode doit être :

```
return convertFilesToProgrammes(Database.getProgrammesByXXX(CLE));
```

2 Préparation des données pour le front-end

Régulièrement, et au moins à chaque actualisation de notre base de données avec un nouveau fichier XMLTV, il faut envoyer les informations au front-end, qui est hébergé sur un autre serveur. La partie front-end est gérée par un autre stagiaire, et nous avons choisis le JSON pour transmettre les informations. Comme le serveur back-end n'est pas visible depuis l'extérieur (pour l'instant), c'est lui qui va s'occuper d'envoyer toutes les informations et ne pourra pas en recevoir.

2.1 Parseur XML

Comme nos fichiers représentant nos chaînes et nos programmes sont en XML, il faut les reconvertir en objets Java avant de les passer en JSON. Ces nouveaux objets ne sont pas les mêmes

que les précédents objets Programme et Channel puisqu'ils ne contiennent pas nécessairement les mêmes informations. En effet, les nouveaux objets (que l'on va appeler SendProgramme et SendChannel) ne contiennent que les informations nécessaires à la partie front-end.

Pour s'occuper du parsage XML vers objets, on va réutiliser des parseurs SAX. Cette fois-ci, il y a un parseur spécifique pour les chaînes et un autre pour les programmes (étant donné qu'ils sont dans des fichiers différents).

2.2 Parseur JSON

Ensuite il faut s'occuper de convertir nos objets vers les fichiers JSON correspondants. Il y a un fichier JSON regroupant toutes les chaînes et plusieurs pour les programmes. Chacun de ces fichiers contient donc soit un tableau JSON de chaînes ou un tableau JSON de Programmes.

L'écriture des fichiers JSON est faite grâce à une classe appelée JsonBuilder. Une instance de cette classe contient un ArrayList d'Object, ainsi qu'une méthode add() pour ajouter un objet au tableau et une méthode buildFile() pour écrire le tableau dans un fichier Json. Pour convertir l'objet en Json, la librairie de Google Gson a été utilisée. Elle permet de convertir directement un objet en une String Json.

2.3 La classe JSONConverter

Un fois que tous les parseurs sont créés, nous avons besoin d'une classe qui s'occupe de les appeler correctement. Cette classe s'appelle la classe JSONConverter. C'est cette classe qui appelle les bonnes méthodes des parseurs sur les bons fichiers pour pouvoir convertir nos XML en JSON.

Répartition des fichiers

Comme dit précédemment, il n'y a qu'un seul fichier JSON qui regroupe toutes les chaînes, en revanche, c'est légèrement plus compliqué pour les programmes. En effet les programmes sont divisés dans différents fichiers JSON en fonction de l'heure à laquelle ils passent. Cela permet à la partie front-end (qui dispose de moins de puissance de calculs) d'avoir moins de traitements à effectuer. Les fichiers JSON regroupent donc les programmes en fonction de leur disponibilité, et sera nommé en fonction du moment de la journée. Par exemple, le fichier allDay.json regroupe tous les programmes passant toute la journée (c'est-à-dire de 0h00 à 23h59) et le fichier evening.json regroupe uniquement les programmes du soir (de 20h00 à 22h59). On peut faire ce regroupement grâce à l'objet Disponibility qui possède une heure de début, une heure de fin, et un nom. Le fichier créé correspondant à la disponibilité aura pour nom le nom de la disponibilité et contiendra tous les programmes qui commencent entre l'heure de début et celle de fin. De plus on veut pouvoir faire cela pour plusieurs jours. Ainsi, on se retrouve avec plusieurs dossiers, chacun représentant un jour (et dont le nom du dossier est la date de ce jour), et dans ce dossier, plusieurs fichiers JSON correspondant à différents moment de la journée.

C'est la classe JiSONConverter qui s'occupe de répartir correctement les programmes dans les bons fichiers et dossiers.

Fichier de configuration

Il est possible de modifier plusieurs paramètres grâce au fichier de configuration du crawler. Les paramètres correspondant au parsing et à l'envoi des JSON sont sous la balise <parser>. Voici les différents paramètres :

— XML.path.channelsFolder et XML.path.programmesFolder : Il s'agit des chemins où seront stockés nos fichers XML contenant les chaînes et les programmes.
— JSON.path.channelsFolder et JSON.path.programmesFolder : Il s'agit des chemins où seront stockés nos fichiers JSON contenant les chaînes et les programmes, avant des les envoyer au front-end.
— JSON.numberOfDays : Défini le nombre de jours pour lequel on créera les JSON. Par exemple s'il est défini sur 2, les JSON seront créés pour la date actuelle et le lendemain.
— JSON.disponibilities : Sous cette balise, il est possible d'ajouter, d'enlever ou de modifier toutes les disponibilités des programmes. Chaque disponibilité entraînera la création d'un fichier JSON correspondant par jour. Les disponibilités, à ajouter dans une balise <disponibility>, possèdent trois paramètres, l'heure de début ("begin"), de fin ("end"), et un nom ("name").

3 Surveillance

La robustesse de l'application est quelque chose de très important. En effet, si l'application crashe, ce sont les utilisateurs qui seront directement impactés. Il a donc fallu mettre en place plusieurs systèmes pour s'assurer que le fonctionnement de l'application est normal, et de pouvoir le corriger si ce n'est pas le cas. Pour cela, nous avons dû implémenter du thread monitoring ainsi que des watchdogs. Ces fonctionnalités sont utilisables à travers les méthodes statiques de la classe Monitoring.

1 Thread monitoring

1.1 Intérêt

Dans notre application, les différents éléments qui peuvent potentiellement crasher (comme les crawlers) sont exécutés au sein de différents threads. Il a donc fallu mettre un place un système pour pouvoir surveiller ces threads. La difficulté ici est de trouver un système qui soit simple à utiliser et non-envahissant et qui permette de recueillir le maximum d'informations sur les différents threads exécutés par l'application. On voudrait ne pas avoir à modifier le code des classes de chaque crawler que l'on souhaite monitorer car cela aurait pour effet de rendre le code plus compliqué et difficilement maintenable.

1.2 Le Monitor

Le Monitor est un objet implémentant l'interface Runnable et qui s'occupe de surveiller tous les threads souhaités. Pour cela il utilise les propriétés liées au thread pour connaître leur état. Ainsi, dès lors qu'un thread change d'état, le Monitor va écrire les informations concernant ce thread dans un fichier JSON. Ces informations sont l'id et le nom du thread, son état, la date et l'heure ainsi que le temps CPU utilisé par le thread et le temps écoulé depuis sa première activation. Une des utilisation finale de ces données serait de les envoyer à la partie front-end pour pouvoir en faire une représentation graphique en temps réel.

Fonctionnement

Le fonctionnement du Monitor se fait grâce au ThreadMXBean de Java, que l'on obtient grâce à une méthode de la classe ManagementFactory. Cet objet nous permet d'obtenir des informations sur les threads actifs de la JVM.

Configuration

Un fichier de configuration regroupant tout ce qui à un lien avec les monitors et les watchdogs existe. Le seule paramètre que l'on peut changer pour le Monitor est le chemin du fichier JSON dans lequel nos informations seront stockées.

2 Watchdogs

Un watchdog en programmation est un objet qui s'assure que des éléments d'une application ne restent pas bloqués. Si l'élément surveillé est bloqué, alors le watchdog le redémarre.

N'ayant pas réussi à trouver de compromis entre facilité d'utilisation et fonctionnalité, deux types de watchdogs ont été implémentés. Le premier est un watchdog assez robuste mais nécessitant de modifier le code de l'objet à surveiller pour pouvoir l'utiliser. Le deuxième est un watchdog bien plus simple à utiliser mais ne proposant pas toutes les fonctionnalités du premier.

2.1 Extern Watchdog

L'ExternWatchdog correspond au premier type de watchdog dont on a parlé plus haut. Il est capable de détecter si un thread reste bloqué pendant un certain temps et de le relancer le cas échéant. Son fonctionnement reste simple, le thread surveillé doit appeler régulièrement une fonction pour informé le Watchdog qu'il est bien actif. Le watchdog de son côté utilise un timer pour chaque thread qu'il surveille. Lorsque que le thread appelle la fonction de réveil, le timer est reset au temps de départ. Si le timer atteint zéro, cela veut dire que le thread n'a pas donné de nouvelle depuis trop longtemps, qu'il est probablement bloqué, et qu'il faut le relancer.

2.1.1 Beta Extern Watchdog

Pour garantir plus de robustesse, une deuxième ExternWatchdog (appelé BetaExternWatchdog) est utilisé. Celui-ci n'a pour rôle que de surveiller ce premier watchdog et de le relancer s'il crashe ou bloque. L'ExternWatchdog principal surveille aussi ce BetaExternWatchdog.

2.1.2 Inconvénient et utilisation

Le défaut majeur de ce thread est qu'il faut modifier le code de l'objet que l'on veut surveiller. Cet objet, tout comme pour le monitoring, doit à l'origine implémenter l'interface Runnable, mais pour l'utiliser avec l'extern Watchdog, doit cette fois-ci étendre la classe abstraite Stoppable et mettre le contenu de ce qui était à la base dans la fonction run() dans la nouvelle fonction à implémenter start(). De plus, il faut appeler dans la boucle principale de l'objet la fonction Monitoring.wakeUp() et passer en argument Thread.currentThread() (le thread actuel). C'est cette fonction qui prévient le watchdog que le thread est actif. Enfin, il faut rajouter dans toutes les boucles susceptibles d'être bloquantes (!isStopped()). C'est cela qui permet au watchdog

d'arrêter correctement le thread. Un fois que ces modifications ont été effectuées, il ne reste plus qu'à lancer l'objet Stoppable avec la fonction Monitoring.addToExternWatchdog(), en passant en argument l'objet Runnable, le nom du futur thread et le temps maximum que le watchdog doit attendre avant de considérer que le thread est bloqué.

Comme on peut le constater, l'utilisation de ce watchdog est assez lourde. Il existe donc un autre watchdog, plus léger qui peut aussi être utilisé.

2.2 Watchdog

Le watchdog le plus simple à utiliser, appelé sobrement Watchdog, est celui utilisé actuellement dans l'application pour surveiller le thread du crawler XMLTV ainsi que celui du monitoring. Son utilisation plus simple implique qu'il est un peu moins robuste. Au lieu d'attendre un message des threads surveillés pour savoir s'il fonctionne, celui-ci regarde l'état de ces threads. S'ils sont arrêtés, cela veut dire qu'il faut les relancer. Le désavantage de cette méthode est que si le thread surveillé boucle indéfiniment, le Watchdog continuera de considérer qu'il est en vie et fonctionne normalement.

2.2.1 Watchdog Beta

Comme pour l'ExternWatchdog, un BetaWatchdog est implémenté pour plus de robustesse. Le BetaWatchdog surveille le Watchdog "normal", le relançant si celui-ci crashe. Le BetaWatchdog fait aussi parti de la liste des threads que le Watchdog "principal" surveille.

2.2.2 Utilisation

L'utilisation du Watchdog est très simple, elle se fait au travers de la classe Monitoring. Il suffit en effet d'avoir un objet implémentant l'interface Runnable et de le lancer automatiquement en appelant la fonction Monitoring.launchMonitorAndWatchThread(), en passant l'instance du Runnable ainsi que le nom que l'on veut donner au thread. Cette fonction a pour effet de lancer la classe dans un nouveau thread qui sera surveillé par le Monitor et le Watchdog. Aucune modification du code n'est donc à faire.

2.3 Configuration

La configuration des watchdogs se fait via le fichier de configuration monitoring_config.xml.

Sous la balise watchdogs, on a :

- watchdog.logsFilePath : Le chemin vers le fichier qui contient les logs du watchdog.
- watchdog.timeOfSleep : Le temps au bout duquel le watchdog vérifie que les threads sont toujours en vie.
- betaWatchdog.logsFilePath : Le chemin vers le fichier qui contient les logs du betaWatchdog.
- betaWatchdog.timeOfSleep : Le temps au bout duquel le betaWatchdog vérifie que le Watchdog est toujours en vie.
- externWatchdog.logsFilePath : Le chemin vers le fichier qui contient les logs de l'externWatchdog.

4 Utilisation du .jar

Un fichier .jar compilé est déjà disponible. Il est possible d'exéctuter ce fichier en utilisant la commande `java -jar -Dfile.encoding=UTF-8 Crawler.jar monitoring_config.xml crawler_config.xml` en mode Administrateur.

Un fichier appelé RUN.bat est présent et permet de ne pas avoir à taper la commande. Il suffit simplement de l'exécuter en mode admin pour lancer l'application.

Descriptif des paramètres

Le paramètre `-Dfile.encoding=UTF-8` permet d'encoder les fichiers écrits en UTF-8, ce qui est important car sinon, les fichiers XML auront du mal à être lus par le parser. C'est aussi à cause de ce paramètre que l'on doit lancer la commande en mode Administrateur car sans cela, les droits d'accès ne sont pas suffisants pour changer ce paramètre.

Ensuite, `Crawler.jar` est le nom du fichier .jar contenant le code compilé, `monitoring_config.xml` est le fichier de configuration du monitoring et des watchdogs, et `crawler_config.xml` est celui du crawler.

Rappelons que les deux fichiers de configuration doivent être présents dans le répertoire contenant le fichier .jar et que tous les dossiers contenus dans les chemins indiqués dans les fichiers de configuration doivent exister.

5 Les fonctionnalités non implémentées

Certaines des fonctionnalités des objectifs initiaux n'ont pas été implémentées, notamment la collecte d'image et la synchronisation FTP.

1 Extension collecte d'images

Initialement, il était prévu d'implémenter un crawler supplémentaire, qui aurait pour objectif de récupérer des images sur internet en lien avec les programmes TV diffusés, un peu à la manière du crawler de Louis. Ce crawler aurait fait parti de ceux qui ajoutent des informations aux fichiers XML contenant les programmes. Comme cette fonctionnalité aurait nécessité énormément de temps de développement, j'ai préféré me concentrer sur le parsage du fichier XMLTV.

2 Synchronisation FTP

Une autre fonctionnalité demandée était la synchronisation FTP avec le front-end. En effet, il avait été choisis d'envoyer les fichiers JSON au front-end en utilisant le protocole FTP. Malheureusement, celui-ci nécessite un serveur FTP du côté du front-end, et le stagiaire s'occupant de cette partie n'a pas eu le temps de s'en occuper. La synchronisation FTP à donc été reportée.

Conclusion

Je suis très satisfait de ces deux mois de stage. Nous avions une grande autonomie et cette liberté m'a permet d'expérimenter différentes méthodes, notamment au niveau du monitoring et de l'optimisation des parseurs.

J'ai pu mettre en pratique mes connaissance en Java, apprendre de nouvelles choses, et mieux maîtriser l'IDE que j'ai utilisé, qui est Eclipse.

Les différentes tâches effectuées sont donc le maintien de la base de données d'images de Louis, l'ouverture du serveur LIFAT, la mise en place du crawler XMLTV configurable, le parseur XMLTV et JSON, ainsi que tout le système de monitoring et de watchdogs.

Annexes

A Fichiers de configurations

Attention, pour tous les champs des fichiers de configuration qui indiquent des emplacements (path), il est nécessaire que les dossiers indiqués dans ces paths existent préalablement.

1 Configuration du crawler : crawler_config.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<config>
  <website>
    <URL>
      <!--The home page of xmltv.fr website-->
      <home>https://www.xmltv.fr</home>

      <!--The link to the XMLTV file-->
      <file>https://www.xmltv.fr/guide/tvguide.xml</file>
    </URL>
  </website>
  <crawler>
    <!--The time between two activations of the crawler (in minutes) -->
    <refreshTime>60</refreshTime>

    <!--Forces the crawler to download the file even if the file is up to date -->
    <forceRecovery>false</forceRecovery>

    <!--The path where the XMLTV file is stored -->
    <XMLTVFile>
      <path>C:/Temp/XMLTV/xmltv.xml</path>
    </XMLTVFile>
  </crawler>
  <parser>
    <filter>
      <enabled>true</enabled>
      <authorizedChannels>
        <!-- TF1 -->
        <channelId>C192.api.telerama.fr</channelId>

        <!-- France 2 -->
```

```
33          <channelId>C4.api.telerama.fr</channelId>
34
35          <!-- France 3 -->
36          <channelId>C80.api.telerama.fr</channelId>
37
38          <!-- Canal+ -->
39          <channelId>C34.api.telerama.fr</channelId>
40
41          <!-- France 5 -->
42          <channelId>C47.api.telerama.fr</channelId>
43
44          <!-- M6 -->
45          <channelId>C118.api.telerama.fr</channelId>
46        </authorizedChannels>
47      </filter>
48      <multithreading>
49        <threadsNumber>16</threadsNumber>
50
51        <!--Number of tasks to compute before continuing parsing -->
52        <tasksNumber>512</tasksNumber>
53      </multithreading>
54      <XML>
55        <path>
56          <channelsFolder>C:/Temp/XML/Channels</channelsFolder>
57          <programmesFolder>C:/Temp/XML/Programmes</programmesFolder>
58          <serializedMaps>C:/Temp/SerializedData/Hashmap</serializedMaps>
59        </path>
60      </XML>
61      <JSON>
62        <path>
63          <channelsFolder>C:/Temp/JSON/Channels</channelsFolder>
64          <programmesFolder>C:/Temp/JSON/Programmes</programmesFolder>
65        </path>
66        <numberOfDays>5</numberOfDays>
67        <disponibilities>
68          <disponibility>
69            <name>allDay</name>
70            <begin>00</begin>
71            <end>24</end>
72          </disponibility>
73          <disponibility>
74            <name>morning</name>
75            <begin>00</begin>
76            <end>12</end>
77          </disponibility>
78          <disponibility>
79            <name>afternoon</name>
80            <begin>12</begin>
81            <end>20</end>
82          </disponibility>
83          <disponibility>
84            <name>evening</name>
85            <begin>20</begin>
86            <end>23</end>
87          </disponibility>
88        </disponibilities>
89      </JSON>
90    </parser>
91  </config>
```

Description

Ce fichier sert à paramétrer tout ce qui est en lien avec la récupération et le parsage du fichier XMLTV.

Sous website.URL :

- home : La page d'accueil du site xmltv.fr, elle sert à savoir si le fichier XMLTV que l'on a déjà téléchargé est toujours à jour ou non en en extrayant la date de mise à jour du fichier.
- file : Le lien vers le téléchargement du fichier xmltv.

Sous crawler :

- refreshTime : La durée de mise en pause du crawler après une tentative de téléchargement. Étant donné que la date de mise à jour présente sur le site xmltv.fr n'est précise qu'à la journée près, il n'est pas nécessaire de fixer le refreshTime très bas. On peut par exemple faire une mise à jour toutes les 24 x 60 minutes. Toutefois si l'on veut espérer télécharger le fichier le plus vite possible après son actualisation, il est recommander de mettre un chiffre plus bas (voir Section 0.1 (Chapitre 2)).
- forceRecovery : Pour les besoins du développement, il a été mis en place un mode où la date de téléchargement n'est plus prise en compte et où le crawler télécharge systématiquement le XMLTV même s'il est déjà à jour. Pour activer ce mode, il suffit de passer ce paramètre à true (voir Section 0.1 (Chapitre 2)).
- XMLTVFile.path : Il s'agit de l'emplacement où sera sauvegardé le fichier XMLTV.

Sous parser.filter (voir Section 0.4.3 (Chapitre 2)) :

- enabled : Active ou desactive le mode filter. Ce mode permet de ne travailler qu'avec certaines chaînes, réduisant considérablement la tailles des données à traiter, à stocker et à envoyer au front-end, ainsi que le temps de traitement.
- authorizedChannel : Liste des identifiants de toutes les chaînes acceptées, à mettre dans une balise <channelId>. Les identifiants sont à aller chercher directement dans les chaînes décrites dans le fichier XMLTV.

Sous parser.multithreading (voir Section 0.4.2 (Chapitre 2)) :

- threadsNumber : Le parsing du fichier XMLTV est fait en multithreading, ce paramètre permet donc d'indiquer le nombre de threads que l'on veut utiliser. Il est recommandé de mettre un nombre de coeurs supérieur ou égal au nombre de coeurs de la machine sur laquelle l'application tourne sans toutefois mettre un nombre excessivement élevé.
- tasksNumber : Ce paramètre est à fixer pour éviter de consommer trop de mémoire. Un nombre trop petit ralentira l'exécution et un nombre trop grand augmentera la mémoire utilisée.

Sous parser.XML.path (voir Section 1 (Chapitre 2)) :

- channelsFolder : Emplacement des fichiers XML contenant les chaînes.
- programmesFolder : Emplacement des fichiers XML contenant les programmes.
- serializedMaps : Emplacement des fichiers .dat serializés contenant les hashmaps.

Sous parser.JSON (voir Section 2 (Chapitre 2)) :

- path.channelsFolder : Emplacement des fichiers JSON prêt à l'envoi au front-end et contenant les chaînes.
- path.programmesFolder : Emplacement des fichiers JSON prêt à l'envoi au front-end et contenant les programmes.
- numberOfDays : Nombre de jours sur lesquels les fichiers JSON doivent être fait, en partant de la date actuelle.
- disponibilities : Liste de créneaux horaires. Les programmes seront regroupés dans des fichiers JSON en fonction des créneaux selon lesquels ils seront diffusés. Chaque créneau contient 3 éléments :

— name : le nom du créneau, ce sera aussi le nom du fichier JSON correspondant à ce créneau.
— begin : L'heure de début du créneau.
— end : L'heure de fin du créneau. Dans la pratique, l'heure de fin utilisée par le programme sera une minute avant celle marquée en paramètre. Par exemple, Si l'on passe 24 en paramètre, l'heure de fin effective du créneau sera 23h59.

2 Configuration du monitoring : monitoring_config.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>

<config>
  <monitoring>
    <!-- The file where all the informations about the threads are stored -->
    <path>C:/Temp/Monitoring/MonitoringData/Monitoring.json</path>
  </monitoring>

  <watchdogs>
    <watchdog>
      <logsFilePath>C:/Temp/Monitoring/Watchdog/logs.txt</logsFilePath>
      <!-- Seconds -->
      <timeOfSleep>3</timeOfSleep>
    </watchdog>
    <betaWatchdog>
      <logsFilePath>C:/Temp/Monitoring/Watchdog/logs.txt</logsFilePath>
      <!-- Seconds -->
      <timeOfSleep>10</timeOfSleep>
    </betaWatchdog>
    <externWatchdog>
      <logsFilePath>C:/Temp/Monitoring/Watchdog/logs.txt</logsFilePath>
    </externWatchdog>
  </watchdogs>
</config>
```

Description

Ce fichier sert à paramétrer tout ce qui est en lien avec la surveillance des threads : monitoring et watchdogs.

Sous monitoring.data (voir Section 1 (Chapitre 3)) :

— path : L'emplacement du fichier où seront stockés les logs du monitoring.

Sous watchdogs.watchdog (voir Section 2 (Chapitre 3)) :

— logsFilePath : L'emplacement du fichier des logs du watchdog.
— timeOfSleep : La durée en secondes entre deux vérifications de la liste des threads du watchdog.

Sous watchdogs.betaWatchdog (voir Section 2 (Chapitre 3)) :

— logsFilePath : L'emplacement du fichier des logs du betaWatchdog (peut être le même fichier que pour le watchdog).
— timeOfSleep : La durée en secondes entre deux vérifications du watchdog.

Sous watchdogs.externWatchdog (voir Section 2 (Chapitre 3)) :

— logsFilePath : L'emplacement du fichier des logs de l'externWatchdog (peut être le même fichier que pour le watchdog).

Bibliographie

[1] Louis Babuchon. « Scraping d'image smart pour portail guide media TV/vidéo ». Projet Recherche & Développement. Tours, France : Ecole Polytechnique de l'Université François Rabelais de Tours, 2018-2019.

[2] Ryan Mitchell. *Web scraping with Python : Collecting more data from the Modern Web*. OReilly Media, Inc., 2018.

[3] Jean-François Pillou. *DOM (Document Object Model) et SAX (Simple API for XML)*. Oct. 2008. url : https://www.commentcamarche.net/contents/1329-dom-document-object-model-et-sax-simple-api-for-xml.

Création d'un crawler XMLTV

Confidentiel

Résumé

Ce document est le rapport de stage dont l'objectif était la création d'un crawler XMLTV. Le but était de récupérer les informations nécessaires à l'élaboration d'un guide TV amélioré. Pour cela, il a fallu créer une application en Java permettant de récupérer un fichier XMLTV, de le parser, et de convertir les données au format JSON pour qu'elles puissent être utilisées par le front-end. Il a aussi fallu mettre en place une structure de monitoring et des watchdogs.

Mots-clés

XMLTV, Java, XML, JSON, Monitoring, Watchdog

Abstract

This report is about the creation of a XMLTV crawler. The aim was to get the necessary informations to create an enhanced TV guide. This report deals with the creation of a Java app that retrieves a XMLTV file, parses it, and converts it to JSON files that can be used by the front-end. A monitoring system and watchdogs had to be created to ensure robustness.

Keywords

XMLTV, Java, XML, JSON, Monitoring, Watchdog

Entreprise
Université de Tours

Étudiant
Romain Hérault (DI4)

Tuteur académique
Vincent T'Kindt

Tuteur entreprise
Mathieu Delalandre (Maître de conférence)